MemOpLight: Leveraging application feedback to improve container memory consolidation

Francis Laniel, Damien Carver, Julien Sopena, Franck Wajsburt, Jonathan Lejeune, Marc Shapiro

HAL Id: hal-03065629
https://hal.science/hal-03065629

Submitted on 14 Dec 2020

MemOpLight: Leveraging application feedback to improve container memory consolidation

Francis Laniel*‡, Damien Carver*‡, Julien Sopena*‡, Franck Wajsburt*, Jonathan Lejeune*‡ and Marc Shapiro*‡

‡*INRIA, DELYS Team, Paris, France*
Sorbonne Université, CNRS, LIP6, F-75005 Paris, France

Index Terms—**Linux, container, memory, memory consolidation**

Abstract—**The container mechanism amortizes costs by consolidating several servers onto the same machine, while keeping them mutually isolated. Specifically, to ensure performance isolation, Linux relies on memory limits. These limits are static, despite the fact that application needs are dynamic; this results in poor performance. To solve this issue, MemOpLight uses dynamic application feedback to rebalance physical memory allocation between containers focusing on under-performing ones. This paper presents the issues, explains the design of MemOpLight, and validates it experimentally. Our approach increases total satisfaction by 13% compared to the default.**

I. INTRODUCTION

A cloud provider is allowed to *consolidate* the logical servers of different clients on the same underlying machine, thus amortizing cost [1]. Each client has the responsibility to correctly size the virtual resources that they rent in order to ensure that they execute smoothly [2], [3], [4]. However, there is a trade-off. The execution of one logical server should not disturb the others: the logical servers should remain *isolated* from one another. In self-managed clouds, the goal is to maximize, on any single machine, the number of applications that respect their SLO (Service Level Objectives) with the minimal amount of hardware.

To ensure both consolidation and isolation, a common approach is to use Virtual Machines (VM). Unfortunately, a VM is heavyweight and waste resources. Resource transfer between VM is complex and difficult to automate [5], [6]. A recent alternative is the "container", a group of processes with sharing and isolation properties [7], [8], [9], [10]. Containers support security (*e.g.*, a file in one container cannot be accessed by another), ease of deployment (*e.g.*, it is possible to start a container with a simple shell command) and fine-grain resource control (*e.g.*, a container can be pinned to a specific CPU core) [11].

To ensure *memory performance isolation*, *i.e.*, guaranteeing to each container enough memory for it to perform well, the administrator limits the total amount of physical memory that the container's processes can use. If it exceeds its limits, some of its memory will be reclaimed, making it available to others. Notably, the Linux kernel reclaims pages from the file page cache, resulting in a performance decrease in containers that perform I/O [12].

In previous work, we showed that the memory consolidation provided by the limit mechanism is imperfect [13].

Moreover, the limits size are static, and do not adapt to the containers' dynamic behavior. This is a problem, because it is hard to estimate, *a priori*, the required amount of memory satisfying an application's performance objectives [14], [15]. Furthermore, the metrics available to the kernel to evaluate its policies (*e.g.*, frequency of page faults, I/O requests, use of CPU cycles, *etc.*) are not directly relevant to performance. Indeed, the application performance is better characterized by application-level metrics, such as response time or throughput [16].

To solve these problems, we propose a new approach, called the Memory Optimization Light[1] (MemOpLight). It is based on application-level feedback from containers. Our mechanism aims to rebalance memory allocation in favor of unsatisfied containers, while not penalizing the satisfied ones. As a side effect, this improves overall resource consumption while consolidating memory. MemOpLight is intended to be used by cloud providers, to make better use of the underlying infrastructure, while guaranteeing good performance to client tasks. The memory of containers receiving low load can be reclaimed, in order to improve performance of these receiving high ones. Our experiments show that MemOpLight increases throughput up to 29.2% compared to the default for the `sysbench` benchmark accessing a `mysql` database.

Our main contributions are the following: (i) The design of a simple feedback mechanism, from application to kernel. (ii) An algorithm for adapting container memory allocation. (iii) An implementation in Linux and its experimental evaluation. We organize the remainder of our paper as follows. Section II presents some technical background. Section III summarizes the issues with existing Linux mechanisms. Section IV presents MemOpLight. Section V compares its performance with existing Linux mechanisms. In Section VI, we review related work. Finally, we conclude and discuss future work in Section VII.

II. TECHNICAL BACKGROUND

Containers are based on the Linux `cgroup` structure for grouping a number of processes [17], [18], [19], [20]. A `cgroup` limits the collective resource usage of its processes. Particularly, the total amount of physical memory used across all the processes of a `cgroup` is capped by the `max` and `soft`

---

[1]Our solution uses the colors of traffic lights to indicate container performance.

limits, which we describe later. In the rest of this section, we study how Linux manages the memory of containers under *memory pressure*, *i.e.* when free physical memory is scarce.

A. No limits

When memory pressure occurs, and if the `max` and the `soft` limits are not set, memory is reclaimed from all containers. For instance, if the memory requirements of some container decreases, the kernel reallocates unused memory to another container. This constitutes memory consolidation. Unfortunately, as Linux allocates unused memory to the file cache, it does not enforce memory performance isolation [12]. Therefore, I/O of one container can impede another container's performance without increasing the performance of the former. In summary, when no limits are set, memory consolidation occurs, at the expense of memory performance isolation.

B. The `max` and `soft` limit mechanism

To solve this problem, Linux provides the `max` and the `soft` limits to guide the memory reclamation of containers. By default, these limits are unset; it is up to the user to set them through `sysfs`. A container's memory footprint cannot exceed its `max` limit. Even if there are free pages, the kernel will not allocate them to a container that has reached its `max` limit. If a process needs physical memory, and its container has already reached its `max` limit, the kernel may reallocate memory from another process of the same container, but not from another container. This mechanism ensures isolation, by avoiding that some container would starve the others.

The `soft` limit is similar, except that it is active only when there is memory pressure. A container's allocation may occasionally exceed its `soft` limit if there is free physical memory. To guarantee good performance under memory pressure, the administrator should set the `soft` limit to approximate the container's Working Set (WS) size [21]. Unfortunately, it is hard to estimate WS size [14], [15].

To summarize, container memory footprint is limited by the `soft` limit when memory pressure occurs, and by its `max` limit at all times. However, these settings are not dynamically related to the containers' current memory needs; they may be higher (impeding memory consolidation) or lower (impeding memory performance isolation).

III. MEMORY CONSOLIDATION *VS.* MEMORY PERFORMANCE ISOLATION

In this Section, we recapitulate our previous experiment showing that containers do not ensure memory performance isolation and memory consolidation at the same time [13].

To highlight this problem, we stressed a `mysql` database, widely used in industry, with a dynamic load generated by `sysbench`, an OLTP benchmark well-known in research [22], [23]. We first run a reference experiment with a single container, to establish a baseline for the ideal performance and resource consumption of the application. Then, we run the same experiment with two containers using different limits configurations. We compared their performance and resource

TABLE I: Summary of memory consolidation and memory performance isolation of existing mechanisms in Linux

Mechanism used	Memory consolidation	Memory performance isolation
limits not set	weak	no
max limit	no	yes
soft limit	no (under memory pressure)	yes

consumption to the baseline. The containers' memory needs fluctuated over time due to variation in received load.

When the limits are not set, memory consolidation occurs, but imperfectly and there is no memory performance isolation. The `max` limit ensures memory performance isolation, but thwarts memory consolidation. The `soft` limit is similar under memory pressure, but allows memory consolidation when memory abounds. We conclude from this experiment that existing mechanisms are insufficient as they do not ensure memory consolidation and memory performance isolation. Table I summarizes these results.

Furthermore, the `max` and `soft` limit are static and thus cannot adapt to dynamic behaviors, especially when the application has a low level of activity, as opposed to being completely stopped [24], [25], [26], [27].

IV. MEMORY OPTIMIZATION LIGHT (MEMOPLIGHT)

A. MemOpLight's functioning

It is difficult to do better with the static limits and the limited information available to the kernel. The kernel only has access to low-level metrics such as the CPU cycles used or the I/O bandwidth. Theses are, at best, proxies for application-level Quality of Service (QoS), and are not representative of the needs of modern applications.

Instead, we propose to base memory reclamation on performance metrics perceived by the application. The container itself declares its current level of QoS, according to its own, application-specific metrics (*e.g.* latency for a web server, frame rate for a streaming application, etc.), compared to some performance objectives. We associate a traffic light color to different states of performance:

- GREEN: The container has reached its maximum level and would not benefit from more resources. For instance, a web server has no requests waiting.
- YELLOW: The container is satisfied with its QoS, but it would do better with more resources. For example, a web server has pending requests in a queue but has answered the earlier with sufficiently low response time.
- RED: The container is not satisfied, *e.g.* a web server is not able to answer requests under a certain latency.

Initially a container starts in the RED state.

Our approach is based on three components.

a) User-kernel communication: We develop a communication mechanism between user and kernel, so a container can communicate its state of performance to the kernel. The Linux kernel provides multiple mechanism for this purpose (*e.g.*, syscall, ioctl, sysfs, socket, *etc.*). We choose the `sysfs` because it is less intrusive for the kernel and has a file-oriented API which is easy to use by users.

TABLE II: The different reclamation cases of MemOpLight

Red `cgroups`	Yellow `cgroups`	Result
0	0	No `cgroups` are reclaimed from.
0	≥ 1	Green `cgroups` are reclaimed from.
≥ 1	0	Green `cgroups` are reclaimed from.
≥ 1	≥ 1	Green `cgroups` are reclaimed from first then yellow ones.

b) Probe: A container is equipped with an application probe, which indicates its color, based on some application-specific. The probe could either exploit existing information in the application, or be a separate script that collects information about the application. For `mysql`, we developed a script that reads the database log and analyzes the request latency. In our prototype, the probe compares the transaction latency of `mysql`, which executes inside the container, to some SLO and informs Linux through `sysfs` every second. Specifically, a container with high throughput declares itself green if its throughput equals the SLO and has no transactions waiting. With low throughput, it is green if it handles ≈ 200 t/s. If it cannot respect its SLO, it is red. Otherwise, if it has transactions waiting, it is yellow.

c) MemOpLight: Our third component is an algorithm that executes when memory is scarce. MemOpLight extends the Linux Page Frame Reclaim Algorithm (PFRA) [28]. When a container needs memory, it takes it from the free physical pages. The PFRA activates when a memory threshold is reached, *i.e.* when free physical memory becomes scarce. It aims to reclaim memory and bring the free physical memory above the threshold. Regarding containers, the PFRA has two parts. The first one consists in recycling memory from one container (local reclaim) while the other reclaims memory from all containers (global reclaim). MemOpLight modifies the PFRA behavior and reclaims memory from containers based on their colors, there are multiple cases shown in Table II.

We make two design decisions. The first is to reclaim from both green and yellow containers when there is a red one. By doing so, we hope to maximize the number of satisfied containers quickly. The other is to limit the rate of reclamation to a small fraction of each container's footprint (we chose 2% each second) in order to avoid performance oscillations. Indeed, if a container declares itself as green or yellow, this must mean that its WS fits into its current physical memory allocation.

This mechanism tends to adapt the amount of memory to what is required for each container to be satisfied. Memory reclaimed from one container can be used by other containers to improve their own performance and satisfaction.

Other, more detailed, technical decisions are documented by comments in the code itself. We implemented as a modification of Linux code to invoke the MemOpLight algorithm once per second when there is memory pressure. If MemOpLight fails to reclaim memory, the `soft` limit mechanism activates and so the PFRA. Our modifications amount to ≈ 400 lines of code [29].

B. Probes

The probe can be seen as a MAPE loop which takes places in user space while our modifications to the Linux kernel correspond to another loop which executes in kernel space [30]. In more detail, the probe does three different things:

1) It collects information about the application.
2) It compares this to a SLO, which was provided by container's owner at its startup, to evaluate its performance state.
3) Finally, it communicates the application's performance state as the green, yellow or red colour to the Linux kernel.

For a web server, a probe might simply read the response time from the server's log and compare it to the SLO. If the response time is higher, then the web server performance is declared to be RED, otherwise GREEN. Another simple example is a streaming application: in this case using all three colours is appropriate. If streaming application sustains the SLO frame rate with high video quality, its state of performance is GREEN. If it has to degrade video quality to sustain the SLO, its declares its state as YELLOW. Otherwise, *i.e.* if it cannot respect its SLO, its performance is declared RED. There is a trade-off between precision (which is important in cloud environment where clients are charged for the memory they use) and the overhead of communicating the colour frequently [31], [32].

V. EVALUATION

A. Description of the experiment

The goal of our experiment is to show that MemOpLight ensures both memory performance isolation and memory consolidation as opposed to existing Linux kernel mechanisms [33].

1) Metrics and baseline: Our experiment studies three metrics: throughput, memory footprint and disk I/O. We define throughput as the number of transactions that the benchmark is able to process in one second. A transaction is a set of SQL requests that read from a database. Memory footprint is the amount of memory used by the container during a given one-second period. Disk I/O counts the number of reads done by the container during a second. Throughput is measured from the benchmark output; memory size and disk activity are gathered by `docker`. To compare fairly, we need reference numbers for these metrics. These numbers were collected by running a reference experiment with a single container that executes the scenario of container A as described in Table III.

2) Two-container experiments: Our experiments feature two concurrent containers, A and B, executing an OLTP workload. Both experience changes in activity. To test satisfaction, we arbitrarily set the SLO of container A to processing 1700 t/s and B 1100 t/s. By doing so, we simulate the fact that A bought a better cloud offer than B. The SLO are pictured as colored dashed lines in figures 1a to 4a [34]. When both containers are

TABLE III: Summary of memory consolidation and memory performance isolation of existing mechanisms in Linux

Phases	Container A	Container B
$\varphi_1(h,h)$	high load	high load
$\varphi_2(h,l)$	high load	low load
$\varphi_3(h,i)$	high load	intermediate load
$\varphi_4(l,l)$	low load	low load
$\varphi_5(i,h)$	intermediate load	high load
$\varphi_6(s,h)$	stopped	high load

offered high load, we expect to observe memory performance isolation. When one of the containers receives high load, and the other receives low load or is stopped, we expect memory consolidation to let the former reach the maximum throughput. Accordingly, the experiment goes through six phases, each lasting 180 seconds, as described in Table III.

We use the notation: $\varphi_n(a,b)$ where n is the phase number, a corresponds to the load of container A and b that of container B. Possible values for a and b are h (high), l (low), i (intermediate) and s (container is stopped). During phase $\varphi_1(h,h)$, physical memory is smaller than the combined size of the databases, hence memory pressure occurs. The size of the page cache decreases, causing an increase in physical disk I/O, hence a decrease in application throughput. In $\varphi_2(h,l)$, if memory consolidation is effective, A should be able to take memory from B, and increase its throughput. We expect both containers to have low throughput in $\varphi_3(h,i)$, similarly to $\varphi_1(h,h)$. In $\varphi_4(l,l)$, containers should be able to answer all the transactions. $\varphi_5(i,h)$ mirrors $\varphi_3(h,i)$. During $\varphi_6(s,h)$, as A stops completely, B should be able to reach maximum performance.

We run this scenario ten times and compute the mean and standard deviation of throughput, memory and disk I/O every second, plotted in Figure 1 to Figure 4.

3) Experimental environment: Our experimental machine is part of the Grid'5000 testbed [35], [36]. It has two Intel®Xeon®Gold 6130 CPU clocked at 2.1 GHz, with 192 GB of DDR4 memory and an SSD [37]. To create memory pressure, we artificially limit memory size, by running the experiments within a VM restricted to 4 CPU cores and 3 GB of memory.

We use qemu 3.1.0 as the VM hypervisor, docker 19.03.2 and its python library docker-py 4.0.2 to manage containers [38], [39], [40], [41]. Our benchmark is sysbench oltp [23]. We modified the benchmark code in order to be able to vary the throughput [42]. The benchmark reads requests from a mysql 5.7 database [22], [43]. We run our experiments on Linux 4.19 [44].

Our two containers (A and B) run with two cores each. Each one reads a database of 4 GB.

B. Baseline experiment with one container

After running the reference experiment, we measure the maximum throughput to be 2000 transactions per second (t/s). The maximum memory footprint and maximum disk I/O are measured at 2.8 GB and 100 reads per second respectively.

These values are pictured in Figures 1 to 4 as dashed black horizontal lines.

We arbitrarily set "low" throughput to 10% of high throughput, *i.e.*, 200 t/s. Low throughput is depicted as horizontal dotted black lines in Figure 1a to Figure 4a. We also define "intermediate" throughput as 1500 t/s.

C. Conventionnal mechanisms with two containers

Let us briefly interpret figures 1 to 3. A more detailed analysis is available in previous work [13].

Without any limit set, $\varphi_1(h,h)$ of Figure 1a shows that there is no isolation, since containers have the same throughput. In this figure, in $\varphi_2(h,l)$, A increases its performance, because B receives low load, but it does not reach its reference level. During $\varphi_6(s,h)$, as A stops, B is able to increase its memory footprint, as shown in Figure 1b, and so reaches the reference performance.

Figure 2a depicts the situation when the max limit is set to 1.8 GB for A and 1 GB for B. A has better performance than B in $\varphi_1(h,h)$. This can be explained by looking at the same phase in Figure 2b. Indeed, container footprints follow their max limits, so A has more memory than B. Note that, in $\varphi_6(s,h)$, where A is stopped, B cannot increase its performance because its footprint is blocked by its max limit.

The soft limit results are similar to that of max limit. The only difference occurs in $\varphi_6(s,h)$ of Figure 3a, where B reaches reference performance. Indeed, as A is stopped, there is no more memory pressure and B can increase its footprint as shown in Figure 3b.

Table I, presented earlier, summarizes these results.

## D. MemOpLight with two containers

MemOpLight is designed to consolidate memory while ensuring memory performance isolation. To verify this, we run again the experiment this time enabling MemOpLight. We plot the results in Figure 4.

$\varphi_1(h,h)$, $\varphi_3(h,i)$ and $\varphi_5(i,h)$ in Figure 4a follow the same pattern as the same phases in Figure 3a. Figure 4b shows that, like Figure 3b, when both containers receive high or intermediate load, their memory footprints follow their soft limits ($\varphi_1(h,h)$, $\varphi_3(h,i)$ and $\varphi_5(i,h)$). A has then a larger footprint than B, this shows that MemOpLight ensures memory performance isolation. Moreover, in these phases, performance of containers A and B are better than with max and soft limits. On average, container A answers to 1351, 1362 and 1370 transactions per second compared to 1290, 1314 and 1358 with the max limit; this represents an increase of respectively 4.7%, 3.7% and 0.9%. With MemOpLight, container B handles almost 933, 999 and 1036 transactions per second while it processes only 894, 968 and 978 transactions per second with the max limit; resulting in an increase of 4.4%, 3.2% and 5.9%. These increases can be explained because MemOpLight permits converging to a balance where containers stop stealing memory each other.

Like $\varphi_6(s,h)$ in Figure 3b, when A stops completely, there is no more memory pressure, B's footprint grows and permits it to increase its performance to the maximum throughput.
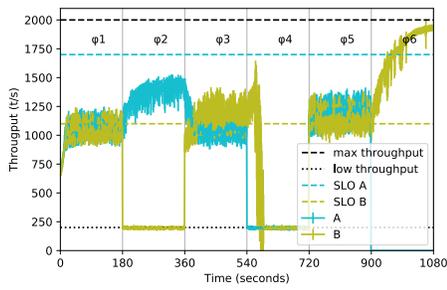
(a) Throughput
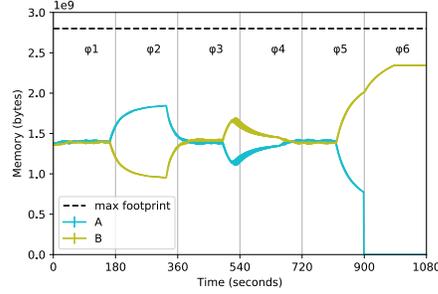
(b) Physical memory footprint
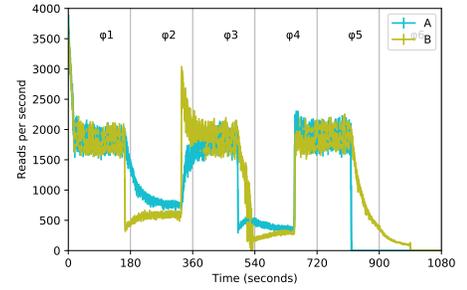
(c) Disk I/O

Fig. 1: No limits set
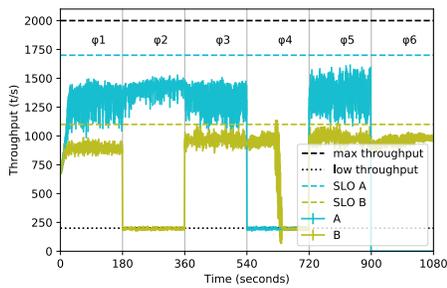
(a) Throughput
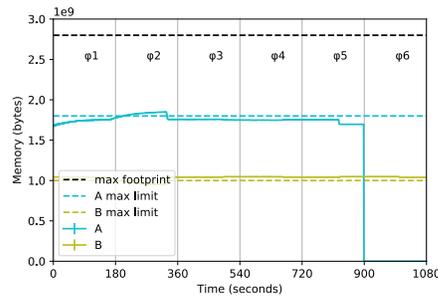
(b) Physical memory footprint
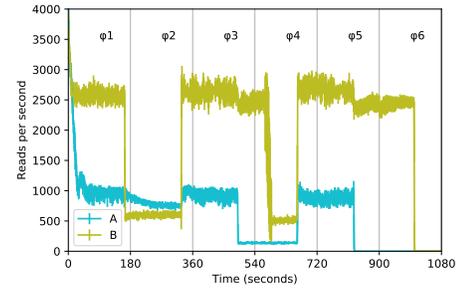
(c) Disk I/O

Fig. 2: Max limits set to $1.8\,\mathrm{GB}$ (A) and $1\,\mathrm{GB}$ (B)
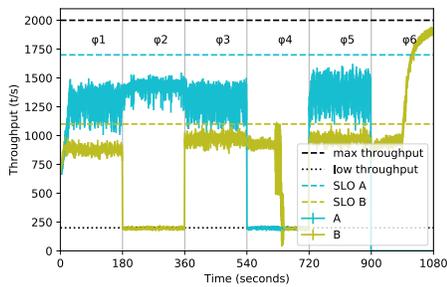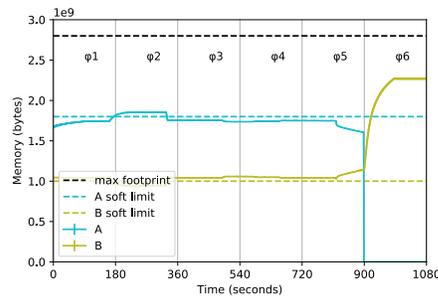
(a) Throughput

(b) Physical memory footprint
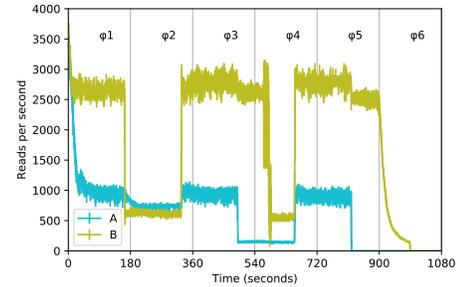
(c) Disk I/O

Fig. 3: Soft limits set to $1.8\,\mathrm{GB}$ (A) and $1\,\mathrm{GB}$ (B)
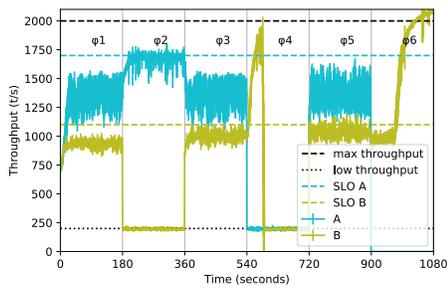
(a) Throughput
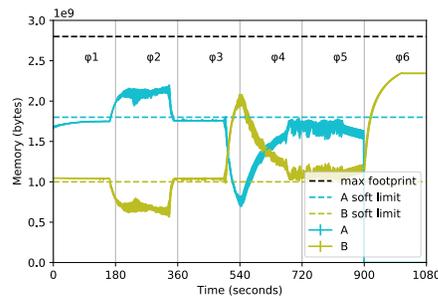
(b) Physical memory footprint
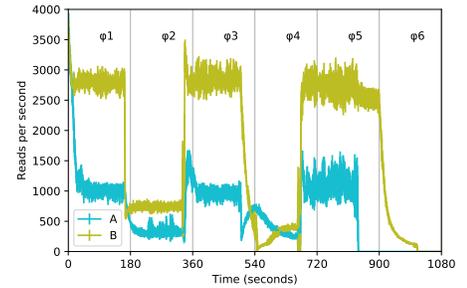
(c) Disk I/O

Fig. 4: MemOpLight with soft limits set to $1.8\,\mathrm{GB}$ (A) and $1\,\mathrm{GB}$ (B)

MemOpLight's dynamic nature brings a real benefit in $\varphi_2(h,l)$. During this phase, B receives low load and A a high one. As shown in Figure 4b, there is a memory transfer from container B to container A. Since container B has no requests in its queue, its memory is reclaimed until finding the threshold where it can still answer to 200 t/s. The application feedback permits finding this memory configuration without previous static analysis. MemOpLight maximizes memory consolidation so container A's throughput equals its SLO.

Table IV summarizes all the previous measurements. B has lower performance with MemOpLight and would have better performance with no limits set. This behavior is normal, since in the latter case there is no memory performance isolation. Nonetheless, this is not expected in cloud since a client could have paid more than another so one client would have a better offer than the other.

In summary, MemOpLight increases performance of containers by redistributing memory following their needs.

E. MemOpLight with eight containers

To confirm that MemOpLight adapts to application loads thanks to application feedback, we now run an experiment with 8 containers. Along the phases, which each lasts for 120 seconds, containers receive high, intermediate or low load. The container loads were chosen randomly to cover different cases and put MemOpLight in difficulty.

The scenario and its characteristics are described in Table V. The first column is SLO. The second one shows the values for max and soft limits. The other columns describe the load at each phase of the experiment. Throughput can have multiple values:

- High: The container receives 2500 t/s. Black cells, in Table V, depicts phases where containers receive high load.
- Intermediate : The container receives its SLO $\pm 5\%$ t/s. In Table V, gray and light gray cells show phases with intermediate load.
- Low: The container sustains only 200 t/s.

The different phases of the scenario can be grouped to form 4 different periods. In φ_1 and φ_2, the system is highly loaded. φ_3 is a transition between highly and lowly loaded. In φ_4, the containers receive low loads. During φ_5 to φ_9, the system is moderately busy but container loads increase over time.

In this experiment, we focus on the respect of SLO and the global performance. Figure 5 plots the average throughput of containers generated across 10 runs for different mechanisms. The global throughput is depicted by the height of the bars while the different colors indicate the throughput of each container. Hatched colors emphasize throughput that is above SLO. Figure 6 focuses on the respect of the SLO. It plots the colors of containers over time during the sixth run of our experiment. To plot this Figure, we activate the probes for all mechanisms. As we discussed in Section IV-B, the probe has no overhead.

In φ_1 and φ_2, the system is overloaded since almost all containers receive high or intermediate loads. The global per-

formance is lower with limits not set, *e.g.* 836453 transactions (t.) compared to 895317.8 t. and 898728 t. for max and soft limits, because containers are fighting for memory so it is used ineffectively. For these phases, MemOpLight achieves the best global performance (927651 t. in φ_1 and 937413 t. in φ_2). Indeed, with MemOpLight, once containers reach a memory balance they stop fighting for memory. Even containers which have less memory have better performance because they use it effectively.

φ_3 is a transition phase between highly and lowly loaded. This phase shows if mechanisms are able to adapt to this change. During this phase, max and soft limits permit better performance than limits not set in φ_3 (779570 t. and 762521 t. compared to 729834 t.). As depicted in Figure 5d, MemOp-Light permits a better global performance (788909), thanks to its dynamic behavior. Figure 6d shows that MemOpLight permits to containers to converge faster to their SLO.

In φ_4, containers continue to receive low loads. In this phase, the max and soft limits activate so they block memory consolidation and impede container performance (583156 t. and 580914 t. compared to 714737 t. without limits set). MemOplight offers the same performance as with limits not set (723084 t. *vs.* 714737 t.). With MemOpLight, all containers are satisfied, since there are only green containers in φ_4 in Figure 6d. It also permits to more containers to exceed their SLO since hatched zones are bigger.

In φ_5 to φ_9, the system is moderately busy but container loads increase over time. Figure 6d shows that MemOpLight maximizes container satisfaction. Moreover, at the beginning of each phase, we can see that containers reach satisfaction more quickly thanks to the dynamic application feedback. During these phases, note that peaks of red can be explained because the benchmark offers an average load, not a constant one.

This experiment shows that MemOpLight permits consolidation and isolation as shown in Table VI and Table VII. Table VI presents performance for each container with all the tested mechanisms. One can see that MemOpLight supports better performance for five of the eight containers. Where MemOpLight performs worse, the difference is only about 0.6%, 2.3% and 1.9% for container B, G and H compared to the experiment with limits not set. Overall, MemOpLight reaches 122.6 millions t. which is 8.9% better than with limits not set. Table VII shows fraction of time where containers are satisfied. This time is the sum of yellow and green time. MemOpLight maximizes the number of satisfied containers. It also increases the time passed satisfied throughout the whole experiment from 44% to 57%. Thus, MemOpLight enhances efficiency, since it improves performance with the same hardware.

In summary, MemOpLight improves throughput and enables containers to be more satisfied.

F. Study of MemOpLight parameters

MemOplight has two parameters that can be tuned: the percent of memory reclaimed each period and the duration

TABLE IV: Median value of measured throughput, averaged over 10 runs, in each phase of each experiment (in t/s rounded to the nearest integer)

Phases / Transactions	$\varphi_1(h,h)$		$\varphi_2(h,l)$		$\varphi_3(h,i)$		$\varphi_4(l,l)$		$\varphi_5(i,h)$		$\varphi_6(s,h)$
Container	A	B	A	B	A	B	A	B	A	B	B
Input load	2000	2000	2000	200	2000	1500	200	200	1500	2000	2000
Limits not set	1053	**1043**	1374	196	1054	**1168**	195	197	1195	**1145**	1751
Max limit	1290	894	1411	196	1314	968	196	660	1358	978	962
Soft limit	1268	879	1423	197	1299	969	196	794	1360	970	974
MemOpLight	**1351**	933	**1670**	195	**1362**	999	196	197	**1370**	1036	1723

TABLE V: Throughput in each phase of experiment (in transactions per second)

Containers	SLO (t/s)	Limit (if set)	φ_1	φ_2	φ_3	φ_4	φ_5	φ_6	φ_7	φ_8	φ_9
A	1800	1400 MB	2500	1710	2500	200	1890	200	200	200	2500
B	1600	1000 MB	1520	2500	2500	200	2500	200	1520	2500	200
C	1400	800 MB	1330	1470	200	1470	2500	1330	1470	200	1470
D	1400	800 MB	2500	1470	200	200	200	1470	200	2500	2500
E	1200	600 MB	1140	1140	200	2500	1260	2500	2500	2500	200
F	1200	600 MB	2500	1260	200	2500	200	2500	1260	1260	2500
G	1000	400 MB	2500	200	2500	200	200	200	200	950	950
H	800	400 MB	840	2500	200	200	760	200	760	760	2500

TABLE VI: Total containers' throughputs (in million of requests with different mechanisms, averaged over 10 runs)

Containers / Mechanism	A	B	C	D	E	F	G	H	Total
No limits set	15.2	14.5	17.3	11.9	18.7	15.9	**8.8**	**10.3**	112.6
Max limit	13.6	13.8	13.8	8.5	14.9	9.9	4.6	6.8	85.9
Soft limit	17.1	**17.4**	17.6	12.6	17.5	13.4	7.3	9.5	112.4
MemOpLight	**17.5**	17.3	**18.4**	**13.7**	**20.2**	**16.8**	8.6	10.1	**122.6**

TABLE VII: Fraction of time (in percent) where container is satisfied (*i.e.* yellow or green) according to mechanisms (averaged over 10 runs)

Containers / Mechanism	A	B	C	D	E	F	G	H	Total
No limits set	42	23	8	35	69	39	62	**71**	44
Max limit	**60**	35	5	37	73	15	53	35	39
Soft limit	56	42	11	35	64	16	53	32	39
MemOpLight	58	**44**	**42**	**52**	**76**	**53**	**67**	67	**57**

of a period. In its implementation, we also chose to reclaim memory from yellow containers, this choice can be discussed.

In this subsection, we first study the impact of these parameters on MemOpLight's performance. For that, we run again the experiment featuring two containers. Then, we modify MemOpLight to create MemOpLightNoYellow where yellow containers are not reclaimed. To compare them, we use again the experiment with eight containers. Unfortunately and due to lack of space, we cannot display the associated figures.

First, we set the reclaim period to be $1\,\mathrm{s}$, and the percent to the following values: 1%, 2%, 5% and 10%. The results for this different values are quite the same. This behavior is due to the using of the PFRA to reclaim memory. This mechanism offers a function which has a memory size to reclaim as parameter. But, this parameter is only indicative, the kernel does whatever it can.

We now fix the memory percent to 2% and use this values for the reclaim period: $1\,\mathrm{s}$, $2\,\mathrm{s}$, $5\,\mathrm{s}$ and $10\,\mathrm{s}$. This time, the results are totally different and a reclaim period of $10\,\mathrm{s}$ shows bad performance. Particularly, for $\varphi_2(h,l)$, a period of $10\,\mathrm{s}$ leads to a median throughput of 1456 t/s compared to the

1638 t/s offered by a period of $1\,\mathrm{s}$. This represents a decrease of 10.6%. So, the reclaim period has to conform with the application activity, if the application reacts quickly to event, the period has to be short, otherwise it can be longer.

To finish, we study the impact of not reclaiming memory from yellow containers by comparing MemOpLight and MemOpLightNoYellow. MemOplight performs better than MemOpLightNoYellow, because containers exceed more their SLO in some phases. More precisely, the averaged total transactions is 122.6 millions for MemOpLight and 118.9 for MemOpLightNoYellow. The global satisfaction also drops from 57% to 52%.

If yellow containers are not reclaimed, there are fewer containers to reclaim, so red containers can less improve their performance. Moreover, yellow containers could need a lot of memory to become green. For these reasons, the yellow state is important and the performance results proved it.

VI. RELATED WORK

Since MemOpLight scales memory according to containers needs, we relate our work to auto-scaling. Auto-scaling con-

sists in adding or removing resources or replicas to follow application needs.

Products with auto-scaling already exist [45], [46], [47], [48], [49]. The majority of the existing approaches use horizontal scaling (*i.e.* add or delete a VM or a container).

Amazon's approach is particularly relevant, because it supports vertical scaling, *i.e.*, giving to or taking resources from a VM or a container [45]. Compared to our solution, Amazon focus on all resources. Allocated resources can be modified dynamically thanks to prediction. This dynamic scaling will add or remove resources to maintain their use at fixed levels. The predictive scaling analyzes up to 14 days of a metric use and is able to make predictions 2 days in the future. The prediction grain is one hour. This auto scaling will add or delete resources to conform with the prediction. If the prediction was too weak the dynamic scaling will be activated. The predictive scaling is based on machine learning. MemOpLight can extend this scaling in case of memory pressure or to increase precision since it activates each second.

In research, auto-scaling also exist [50], [51], [52], [53]. Some use horizontal scaling [50], [52]. Lorido-Botran *et al.* do horizontal scaling for containers [51]. They monitor the requests received by their web application during a given period. They analyze and predict the number of requests to come during the next period. Then, they compute the number of containers needed to face those requests. Our approach can extend theirs by adding memory scaling to horizontal scaling.

Dupont *et al.* base their approach on MAPE-K loop [54]. Their approach is innovative in the way that, when a peak of load occurs, they scale the software running by either add/remove function (*e.g.* security features, logging, *etc.*) or increase/decrease quality of execution (*e.g.* decreasing video quality). Their work can be related to us in the sense that we do not scale horizontally or vertically, we just do better with what we have.

Carver *et al.* focus on containers [55]. Since, under memory pressure, all memory `cgroups` are reclaimed they propose to modify Linux memory reclamation algorithm, to target first the least recently active container. This solution avoids performance drop of active containers because they are not reclaimed anymore. As they do not take into account application feedback, it is possible that a green container is always recently activated and thus, never gives up its memory under their solution. MemOpLight could be used in combination with this solution to decide from which green container the kernel should reclaim memory based on the activity.

Compared to all these works, ours is unique in the way that we ensure container satisfaction. MemOpLight is also original in the way that it bases its reclaim on containers' performance as reported by probes. Our mechanism is implemented into the Linux kernel and profits from a global view of resources usage, compared to user space solutions which only have access to the application resources usage.

VII. CONCLUSION AND FUTURE WORK

In this paper, we showed that current Linux mechanisms do not combine memory consolidation and memory performance isolation. By default, there is no memory performance isolation. Memory consolidation is only partially effective. Memory performance isolation can be achieved, at the expense of memory consolidation, using the `max` and `soft` limits.

We presented MemOpLight to provide memory consolidation. An application probe gives information about a container's satisfaction to the kernel. The kernel reacts by reallocating memory in favor of unsatisfied containers, without overly depriving the others. Evaluation shows that containers are satisfied more often thanks to memory consolidation.

In future work, we will test MemOpLight with a real web server application. We also plan to modify the probe to compute an average on feedback instead of giving an instant one. Another perspective is to be able to estimate the WS size of containers by using an application probe. Indeed, it is possible to reclaim memory from a container until it indicates it has "bad" performance. The memory footprint before the transition from "good" performance to "bad" one equals approximately the WS size. We also plan to scale other resources (*e.g.* CPU time or disk bandwidth) according to containers needs.

REFERENCES

[1] Y. Xing and Y. Zhan, "Virtualization and Cloud Computing," in *Future Wireless Networks and Information Systems*, Y. Zhang, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, vol. 143, pp. 305–312. [Online]. Available: http://link.springer.com/10.1007/978-3-642-27323-0_39

[2] Amazon, "Amazon Compute Service Level Agreement." [Online]. Available: https://aws.amazon.com/compute/sla/?nc1=h_ls

[3] Microsoft, "SLA summary for Azure services." [Online]. Available: https://azure.microsoft.com/en-us/support/legal/sla/summary/

[4] OVH, "CONDITIONS PARTICULIÈRES DU SERVICE OVH PUBLIC CLOUD." [Online]. Available: https://www.ovh.com/fr/support/documents_legaux/contractPartOVHCloudFr.pdf

[5] KVM, "Automatic Ballooning," 2013. [Online]. Available: https://www.linux-kvm.org/page/Projects/auto-ballooning

[6] I. Mammedov, "Features/CPUHotplug," Jan. 2017. [Online]. Available: https://wiki.qemu.org/Features/CPUHotplug

[7] Amazon, "Amazon Elastic Container Service." [Online]. Available: https://aws.amazon.com/ecs/?nc1=h_ls

[8] Microsoft, "Microsoft Web App for Containers." [Online]. Available: https://azure.microsoft.com/en-us/services/app-service/containers/

[9] Alibaba, "Alibaba Container Service." [Online]. Available: https://www.alibabacloud.com/product/container-service?spm=a2c5t.10695662.1996646101.searchclickresult.55a3212bnXyv1v

[10] OVH, "OVH Kubernetes." [Online]. Available: https://www.ovh.com/fr/kubernetes/

[11] Docker Inc., "What is a Container?" [Online]. Available: https://www.docker.com/resources/what-container

[12] The kernel development community, "Concepts overview." [Online]. Available: https://www.kernel.org/doc/html/latest/admin-guide/mm/concepts.html

[13] F. Laniel, D. Carver, J. Sopena, F. Wajsburt, J. Lejeune, and M. Shapiro, "Highlighting the Container Memory Consolidation Problems in Linux," in *2019 IEEE 18th International Symposium on Network Computing and Applications (NCA)*. Cambridge, MA, USA: IEEE, Sep. 2019, pp. 1–4. [Online]. Available: https://ieeexplore.ieee.org/document/8935034/

[14] B. Gregg, "Working Set Size Estimation," Feb. 2018. [Online]. Available: http://www.brendangregg.com/wss.html

[15] V. Nitu, A. Kocharyan, H. Yaya, A. Tchana, D. Hagimont, and H. Astsatryan, "Working Set Size Estimation Techniques in Virtualized Environments: One Size Does not Fit All," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 2, no. 1, pp. 1–22, Apr. 2018. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3203302.3179422

[16] J. Weiner, "PSI - Pressure Stall Information," Apr. 2018. [Online]. Available: https://www.kernel.org/doc/html/latest/accounting/psi.html

[17] K. Hiroyu, "Cgroup And Memory Resource Controller," Nov. 2008. [Online]. Available: https://www.static.linuxfound.org/jp_uploads/seminar20081119/CgroupMemcgMaster.pdf

[18] Rami Rosen, "Namespace and cgroups, the basis of Linux containers," Seville, Spain, Feb. 2016. [Online]. Available: https://www.netdevconf.org/1.1/proceedings/slides/rosen-namespaces-cgroups-lxc.pdf

[19] Linux, "Memory Resource Controller." [Online]. Available: https://www.kernel.org/doc/Documentation/cgroup-v1/memory.txt

[20] Zhenyun Zhuang, Cuong Tran, J. Weng, H. Ramachandra, and B. Sridharan, "Taming memory related performance pitfalls in linux Cgroups," in *2017 International Conference on Computing, Networking and Communications (ICNC)*. Silicon Valley, CA, USA: IEEE, Jan. 2017, pp. 531–535. [Online]. Available: http://ieeexplore.ieee.org/document/7876184/

[21] P. J. Denning, "The working set model for program behavior," in *Proceedings of the ACM symposium on Operating System Principles - SOSP '67*. Not Known: ACM Press, 1967, pp. 15.1–15.12. [Online]. Available: http://portal.acm.org/citation.cfm?doid=800001.811670

[22] "mysql." [Online]. Available: https://dev.mysql.com/doc/refman/5.7/en/

[23] Alexey Kopytov, "sysbench."

[24] L. A. Barroso, J. Clidaras, and U. Hölzle, "The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second edition," *Synthesis Lectures on Computer Architecture*, vol. 8, no. 3, pp. 1–154, Jul. 2013. [Online]. Available: http://www.morganclaypool.com/doi/abs/10.2200/S00516ED2V01Y201306CAC024

[25] L. A. Barroso and U. Hölzle, "The Case for Energy-Proportional Computing," *Computer*, vol. 40, no. 12, pp. 33–37, Dec. 2007. [Online]. Available: http://ieeexplore.ieee.org/document/4404806/

[26] D. Meisner, B. T. Gold, and T. F. Wenisch, "PowerNap: eliminating server idle power," *ACM SIGARCH Computer Architecture News*, vol. 37, no. 1, p. 205, Mar. 2009. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2528521.1508269

[27] H. Liu, "A Measurement Study of Server Utilization in Public Clouds," in *2011 IEEE Ninth International Conference on Dependable, Autonomic and Secure Computing*. Sydney, Australia: IEEE, Dec. 2011, pp. 435–442. [Online]. Available: http://ieeexplore.ieee.org/document/6118751/

[28] D. P. Bovet and M. Cesati, *Understanding the Linux kernel: from I/O ports to process management*, 3rd ed. Beijing: O'Reilly, 2006, oCLC: 255008440.

[29] Francis Laniel, "Thesis_linux," 2020. [Online]. Available: https://gitlab.com/eiffel_thesis/thesis_software/thesis_linux

[30] J. Kephart and D. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, Jan. 2003. [Online]. Available: http://ieeexplore.ieee.org/document/1160055/

[31] Amazon, "Amazon EC2 Pricing," Tech. Rep. [Online]. Available: https://aws.amazon.com/ec2/pricing/on-demand/?nc1=h_ls

[32] Google, "VM instances pricing," Tech. Rep., Feb. 2020. [Online]. Available: https://cloud.google.com/compute/vm-instance-pricing

[33] F. Laniel, "Thesis_experiments," 2020. [Online]. Available: https://gitlab.com/eiffel_thesis/thesis_software/thesis_experiments

[34] B. Beyer, C. Jones, J. Petoff, and N. R. Murphy, *Site reliability engineering: how Google runs production systems*, 2016, oCLC: 930683030.

[35] D. Balouek, A. Carpen Amarie, G. Charrier, F. Desprez, E. Jeannot, E. Jeanvoine, A. Lèbre, D. Margery, N. Niclausse, L. Nussbaum, O. Richard, C. Pérez, F. Quesnel, C. Rohr, and L. Sarzyniec, "Adding virtualization capabilities to the Grid'5000 testbed," in *Cloud Computing and Services Science*, ser. Communications in Computer and Information Science, I. I. Ivanov, M. van Sinderen, F. Leymann, and T. Shan, Eds. Springer International Publishing, 2013, vol. 367, pp. 3–20.

[36] F. Laniel, "Thesis_g5kscripts," 2019. [Online]. Available: https://gitlab.com/eiffel_thesis/thesis_software/thesis_g5kscripts

[37] Intel, "Intel Xeon Processor gold 6130," 2017. [Online]. Available: https://ark.intel.com/content/www/us/en/ark/products/120492/intel-xeon-gold-6130-processor-22m-cache-2-10-ghz.html

[38] Qemu, "Qemu," May 2019. [Online]. Available: https://wiki.archlinux.org/index.php/QEMU

[39] Docker, "Docker CE." [Online]. Available: https://github.com/docker/docker-ce

[40] "docker-py." [Online]. Available: https://github.com/docker/docker-py

[41] F. Laniel, "Thesis_scripts," 2020. [Online]. Available: https://gitlab.com/eiffel_thesis/thesis_software/thesis_scripts

[42] ——, "Thesis_sysbench," 2019. [Online]. Available: https://gitlab.com/eiffel_thesis/thesis_software/thesis_sysbench

[43] ——, "Thesis_images," 2020. [Online]. Available: https://gitlab.com/eiffel_thesis/thesis_software/thesis_images

[44] Greg Kroah-Hartman, "Linux 4.19," Oct. 2018. [Online]. Available: https://lkml.org/lkml/2018/10/22/184

[45] Amazon, "AWS Auto Scaling Guide de l'utilisateur," Amazon, Tech. Rep., Nov. 2018. [Online]. Available: https://docs.aws.amazon.com/fr_fr/autoscaling/plans/userguide/as-plans-ug.pdf#auto-scaling-getting-started

[46] J. Barr, "New AWS Auto Scaling – Unified Scaling For Your Cloud Applications," Jan. 2018. [Online]. Available: https://aws.amazon.com/fr/blogs/aws/aws-auto-scaling-unified-scaling-for-your-cloud-applications/

[47] "Microsoft Azure: Autoscaling," May 2017. [Online]. Available: https://docs.microsoft.com/en-us/azure/architecture/best-practices/auto-scaling

[48] "Google Cloud : Équilibrage de charge et scaling," Dec. 2018. [Online]. Available: https://cloud.google.com/compute/docs/load-balancing-and-autoscaling#policies

[49] "kubernetes/autoscaler," Feb. 2019. [Online]. Available: https://github.com/kubernetes/autoscaler/blob/master/cluster-autoscaler/FAQ.md

[50] M. Kriushanth and L. Arockiam, "Load balancer behavior identifier (LoBBI) for dynamic threshold based auto-scaling in cloud," in *2015 International Conference on Computer Communication and Informatics (ICCCI)*. Coimbatore, India: IEEE, Jan. 2015, pp. 1–5. [Online]. Available: http://ieeexplore.ieee.org/document/7218115/

[51] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano, "A Review of Auto-scaling Techniques for Elastic Applications in Cloud Environments," *Journal of Grid Computing*, vol. 12, no. 4, pp. 559–592, Dec. 2014. [Online]. Available: http://link.springer.com/10.1007/s10723-014-9314-7

[52] L. R. Moore, K. Bean, and T. Ellahi, "Transforming reactive auto-scaling into proactive auto-scaling," in *Proceedings of the 3rd International Workshop on Cloud Data and Platforms - CloudDP '13*. Prague, Czech Republic: ACM Press, 2013, pp. 7–12. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2460756.2460758

[53] J. Rao, X. Bu, C.-Z. Xu, and K. Wang, "A Distributed Self-Learning Approach for Elastic Provisioning of Virtualized Cloud Resources," in *2011 IEEE 19th Annual International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*. Singapore, Singapore: IEEE, Jul. 2011, pp. 45–54. [Online]. Available: http://ieeexplore.ieee.org/document/6005367/

[54] S. Dupont, J. Lejeune, F. Alvares, and T. Ledoux, "Experimental Analysis on Autonomic Strategies for Cloud Elasticity," in *2015 International Conference on Cloud and Autonomic Computing*. Boston, MA, USA: IEEE, Sep. 2015, pp. 81–92. [Online]. Available: http://ieeexplore.ieee.org/document/7312143/

[55] D. Carver, J. Sopena, and S. Monnet, "ACDC: Advanced consolidation for dynamic containers," in *2017 IEEE 16th International Symposium on Network Computing and Applications (NCA)*. Cambridge, MA: IEEE, Oct. 2017, pp. 1–8. [Online]. Available: http://ieeexplore.ieee.org/document/8171363/
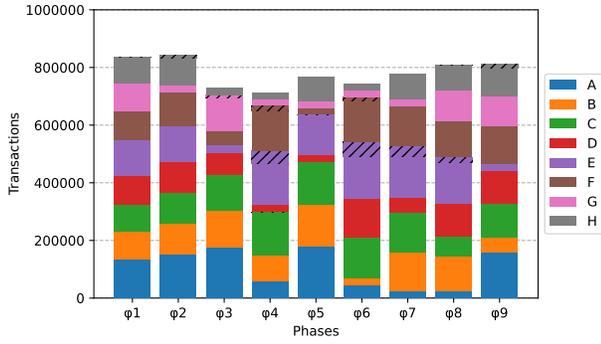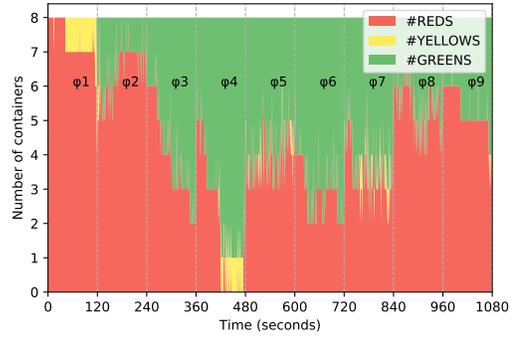
(a) Limits not set
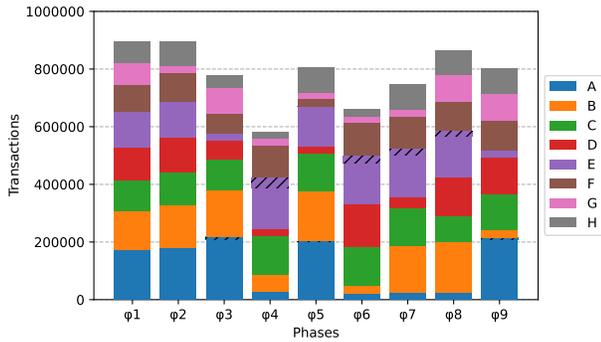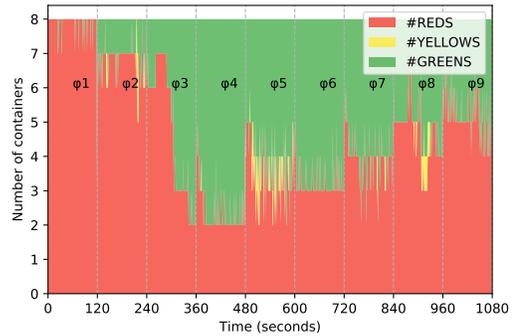
(b) Max limits
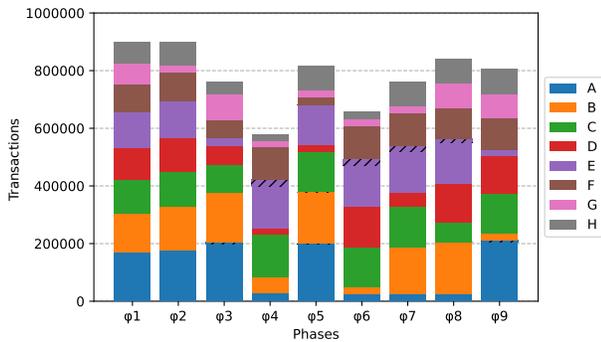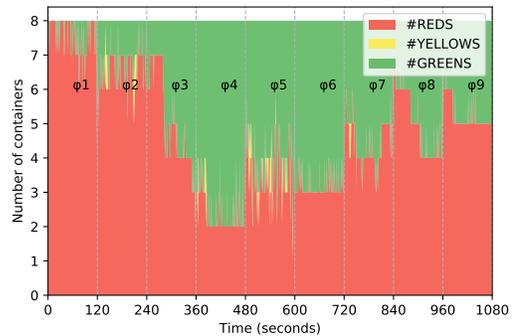

(c) Soft limits
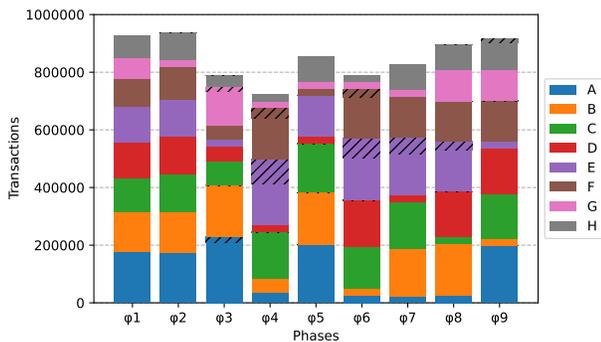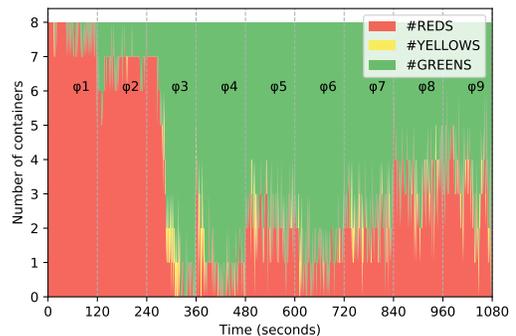
(d) MemOpLight

Fig. 5: Average throughput of eight containers during each phase with different mechanisms


(a) Limits not set

(b) Max limits

(c) Soft limits

(d) MemOpLight

Fig. 6: Containers' colors during the 5th run with different mechanisms